#### SE基础

1. JDK和JDR

JDK开发工具包提供开发环境和运行环境。

JRE java运行环境

JDK包含JRE

1. **== 和 equals**

==比较的是引用

String x = **"abc"**;  
String y = **"abc"**;  
System.***out***.println(x==y); //true

Abc是常量池常量只有一个引用地址

String x = **new** String(**"abc"**);  
String y = **new** String(**"abc"**);  
System.***out***.println(x==y);

new Stirng 在堆上开辟一个新空间指向常量池abc但是在栈上有一个新的引用x所以x和y 的引用不同

两者hashCode相同 也就是说hashCode相等 引用不一定相等

HashCode指的是常量池的hashCode

Equals比较的值，看两者字面量值相等不相等

实际上String equals就是重写Object中的equals方法

1. **haCode（）相等 euqals不一定相等**

hashCode相等只能说明两者hashCode相同 不能说明euqlas相等

Equals相等则hashCode一定相等

1. **final 家族的作用**

Final 可以修饰类 该类不能被继承

可以修饰方法 此方法不能被重写

可以修饰变量 变量必须初始化成为常量不能修改

Finally 异常中出现表示一定执行的代码块 不一定必须写但是写了一定执行

1. **Math方法**

Math.rount() 大于0 遵循四舍五入

小于零 反向四舍五入

System.***out***.println( Math.*round*(5.5)); 6  
System.***out***.println(Math.*round*(5.4)); 5  
System.***out***.println(Math.*round*(-5.5)); -5  
System.***out***.println(Math.*round*(-5.6)); -6

1. **基本的数据类型**

Byte boolean char short int float long double

引用类型 包装类、String

1. **自动拆装箱机制**

**int** x = 10; 装箱  
**int** y = x; 拆箱

1. **String家族**

|  |  |  |  |
| --- | --- | --- | --- |
| String | 操作产生新的字符串 |  |  |
| StringBuilder | 不产生（操作对象） | 不安全 | 单线程性能高 |
| StringBuffer | 不产生（操作对象） | 线程安全 | 多线程使用 |

String str = “i” 与 String str = new String(“i”)

不相同 内存分配不同

String str = “i” 在常量池中产生

String str = new String(“i”) 在堆内存中开辟

常用方法

* indexOf()：返回指定字符的索引
* charAt()：返回指定索引处的字符
* replace()：字符串替换。
* trim()：去除字符串两端空白
* split()：分割字符串，返回一个分割后的字符串数组。
* getBytes()：返回字符串的 byte 类型数组。
* length()：返回字符串长度。
* toLowerCase()：将字符串转成小写字母。
* toUpperCase()：将字符串转成大写字符。
* substring()：截取字符串

1. **抽象类**

抽象类无法直接创建对象 只能被子类继承实例

可以不定义抽象方法

抽象类不能使用final修饰，使用fianl无法被其他类继承

|  |  |
| --- | --- |
| 抽象类 | 接口 |
| abstract | interface |
| 子类 extends | Iplements |
| 可以有构造函数 | 只能有抽象方法 |
| 单继承 | 多实现 |
| 方法权限任意 | 默认public |

1. **Java IO**

Input 输入流 字节 字符

Output 输出流 字节 字符

BIO 同步阻塞IO 处理并发能力低

NIO 同步费组设IO 用过channel实现对路复用

AIO 异步费堵塞IO 异步操作基于事件和回调机制

1. **Files常用方法**

* Files. exists()：检测文件路径是否存在。
* Files. createFile()：创建文件。
* Files. createDirectory()：创建文件夹。
* Files. delete()：删除一个文件或目录。
* Files. copy()：复制文件。
* Files. move()：移动文件。
* Files. size()：查看文件个数。
* Files. read()：读取文件。
* Files. write()：写入文件。

**多线程**

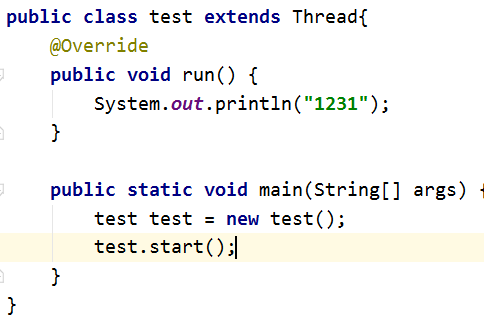
1. 进程和线程

进程 进行资源费配和调度的金本单位

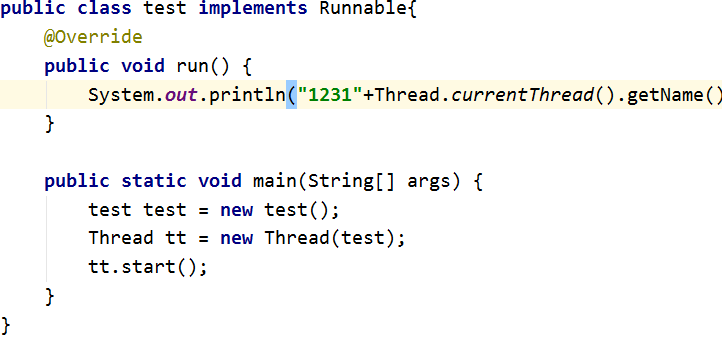
线程 程序流的最小单元

一个进程有多个线程

1. 实现方式
2. 继承Thread 重写run()



（2）实现Runnable接口 覆写run方法



但是没有返回值

（3）对Runable补充实现Callable接口

```javasctipt

**public class** test **implements** Callable<String> {  
 @Override  
 **public** String call() **throws** Exception {  
 System.***out***.println(Thread.*currentThread*().getName()+**""**);  
 **return "1231"**;  
 }  
  
 **public static void** main(String[] args) **throws** ExecutionException, InterruptedException {  
 test test = **new** test();  
 FutureTask<String> task = **new** FutureTask<>(test);  
 Thread tt = **new** Thread(task);  
 tt.start();  
 System.***out***.println(task.get());  
 }  
}

```

有返回值 不能直接传入Thread中 借助FutureTak的方式加载多线程

不再是run方法 call方法

1. 线程池的方式

四种常见的线程池

1. CachedThreadPool 可缓存的线程池 没有核心池 非核心池数量Integer.max\_value 适用于耗时少 任务量大

```javascript

test test = **new** test();  
FutureTask<String> task = **new** FutureTask<>(test);  
Thread tt = **new** Thread(task);  
ExecutorService ee = Executors.*newCachedThreadPool*();  
ee.submit(tt);  
System.***out***.println(task.get());

```

1. FixedThreadPool 固定大小的线程池

```javascript

ExecutorService ee = Executors.*newFixedThreadPool*(4);  
ee.submit(tt);

```

容纳的最大线程数设置的核心线程数

1. ScjeduledThreadPool 核心线程固定 非核心线程没有限制

```javscript

ExecutorService ee = Executors.*newScheduledThreadPool*(4);  
ee.submit(tt);

```

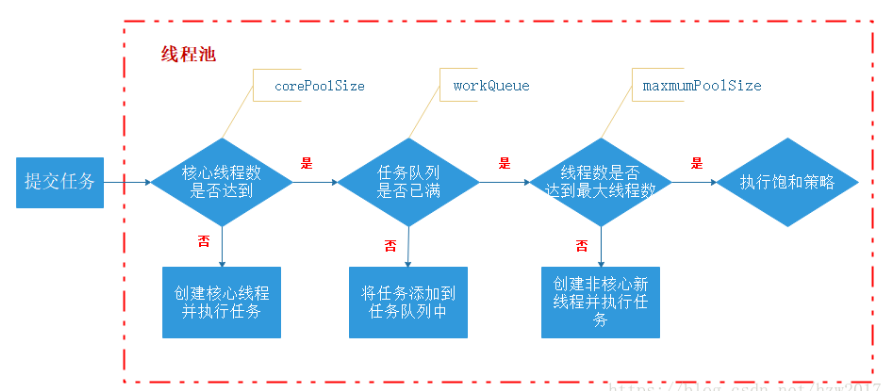
1. SingleThreadExecutor 只有一个核心线程池 对于任务队列没有限制一个任务执行其他任务等待

```javascript

ExecutorService ee = Executors.*newSingleThreadExecutor*();  
ee.submit(tt);

```

线程池的工作流程



1. 提交任务
2. 首先制性判断核心线程池数量是否达到最大值（corePoolSize）未达到最大值创建核心线程 执行任务
3. 核心线程池已满 判断任务队列（workQueue）是否达到最大值未达到最大值将任务添加到认为队列中
4. 任务队列已满 判断线程数（maxmumPoolSize）是否达到最大值未达到最大值创建非核心线程 并提交任务队列
5. 线程数达到最大值 执行handler实现拒绝策略

**线程安全**

多线程条件下保证线程安全 采用加锁机制

**Synchronized**

1. 可以在任意对象以及方法上加锁 倍加锁的代码成为 互斥区 或 临界区
2. 多线程条件下 排队按照CPU分的方式进入同步队列 一个线程执行sunchronize其他线程拿不到 就会出现锁竞争
3. Sychronized实现时是对对象加锁（this 类.class）

**Sychronize实现原理**

1）内置锁和护斥锁

内置锁 每一个继承Object的对象都可以用实现同的锁 线程进入同步代码块同步方法都会获得 该所 退出时释放

互斥锁 内置锁通过互斥所实现 某一个时刻最多只能有一个线程持该锁

实现原理

1. 每一个对象都会有一个monitor
2. 当一个线程获取一个对象 通过monitor监视器就会获取对象的monitor
3. 当这个对象被加锁 线程monitor监视器计数器count++
4. 其他对象因为监视器count不为0 无法进入锁的代码块
5. 当这个对象执行完毕 锁的代码块 monitor监视器计数器count--
6. 当count=0时其他对象都有可能会被获取
7. 调用wait（）方法是 释放当前持有的monitor
8. sychronized锁可重入

在一个被sychronized修饰的代码块中再次添加对象锁sychronized双重加锁如果锁的是同一个对象 就会成功不受阻塞

**实现对多个对象加锁 static修饰**

出现异常锁自动释放

应用单例模式（普通加锁）

```javascript

**class** Singleton{  
 **private static** Singleton *instance* = **null**;  
 **private** Singleton(){  
  
 }  
 **public static synchronized** Singleton newinstance(){  
 **if**(*instance* == **null**){  
 *instance* = **new** Singleton();  
 }  
 **return** *instance*;  
 }  
}

```

保证同一时刻只能有一个线程进入 产生一个对象 线程安全

单例模式（双重校验锁）

```javascript

**class** DubbleSingleton{  
 //**private static** DubbleSingleton *instance*;  
 **private static volatile** DubbleSingleton *instance*;  
  
 **public static** DubbleSingleton getinstance(){  
 **if**(*instance*==**null**){  
 **synchronized** (DubbleSingleton.**class**){  
 **if**(*instance*==**null**){  
 *instance* = **new** DubbleSingleton();  
 }  
 }  
 }  
 **return** *instance*;  
 }  
}

```

采用双重判断加锁的模式 避免多个线程同时进入创建对象

减小锁粒度 锁优化的一种

但是因为JVM虚拟机 会重新编译文件存在 指令重排优化 可能导致初始化丹迪对象将该对象的地址复制给instance字段顺序与上面java中顺序不同

使用volatile 修饰 进制指令重排序优化

**Sychronize锁优化（JDK 1.6之后）**

锁的四种状态 无所状态—>偏向锁状态—>轻量级锁—>重量级锁

偏向锁 ： 乐观锁如果一个线程获取到了锁 那么锁进入偏向模式 当这个锁再次请求锁时 无需在进行任何操作

轻量级锁 ： 当一个线程访问同步资源 锁竞争不激烈 很有可能对此请求来自同一个线程 偏向锁失败 升级成轻量级锁 实质将对象mark-word头部致真 指向线程堆栈 内部 判断是否持有对象

重量级锁： 轻量级锁失败 膨胀成重量级锁

自旋锁 ： 轻量级锁失败后 JVM会进行 自旋锁 让当前线程做空循环 获取则成功不获取 则挂起

**CAS自旋锁**

需要读写的内存值 V 进行比较的值 A 拟写入的新值 B

当V==A 时 CAS 通过原子方式 使用B 更新V值

**锁粗化**

连续的多个srchronize修饰的代码块 且 加锁的对象 是同一个则我们可以将这些代码块合并 采用一个srchroinzed加锁 但是要连续

```javascript

**public class** test {  
 **public void** coarse(){  
 **synchronized** (test.**class**){  
 *//TDDD* }  
 **synchronized** (test.**class**){  
 *//TDDD2* }  
 }

```

合并后

```javscript

**public class** test {  
 **public void** coarse(){  
 **synchronized** (test.**class**){  
 *//TDDD  
 //TDDD2* }  
 }

```

减小开销提高效率

**锁消除**

Jvm 在编译器运行时检测到没必要加锁的内容会自动去除

**线程间通信**

Object类下

Wait() 等待将正在执行的线程书房器质性资格和执行权 不出存在线程池中

notify() 唤醒 唤醒 唤醒鲜橙汁中被wait（） 调用的线程 随机唤醒

NotifyAll() 唤醒全部wait（）等待线程

```javascript

**package** main.ProducerAndConsumer;  
**class** Resource{  
 **private** String **name**;  
 **private int count** = 1;  
 **private boolean flag**;  
  
 **public synchronized void** set(String name){  
 **while**(**flag**){  
 **try**{  
 **this**.wait();  
 }**catch** (Exception e){  
 e.printStackTrace();  
 }  
 }  
 **this**.**name** = name + **count**;  
  
 **count**++;  
  
 System.***out***.println(  
 Thread.*currentThread*().getName()+**"This was producered"** +  
 **":"**+**this**.**name**);  
 **flag** = **true**;  
 **this**.notifyAll();  
 }  
 **public synchronized void** out(){  
 **while**(!**flag**){  
 **try**{  
 **this**.wait();  
 }**catch** (Exception e){  
 e.printStackTrace();  
 }  
 }  
 **count**--;  
 System.***out***.println(Thread.*currentThread*().getName()  
 +**"The consumer is:"**+**this**.**name**);  
  
 **flag** = **false**;  
 **this**.notifyAll();  
 }  
}  
**class** Producer **implements** Runnable{  
 **private** Resource **r**;  
 **public** Producer(Resource r){  
 **this**.**r** = r;  
 }  
 @Override  
 **public void** run() {  
 **while**(**true**){  
 **r**.set(**"商品"**);  
 }  
 }  
}  
**class** Consumer **implements** Runnable{  
 **private** Resource **r**;  
 **public** Consumer(Resource r){  
 **this**.**r** = r;  
 }  
 @Override  
 **public void** run() {  
 **while** (**true**){  
 **r**.out();  
 }  
 }  
}  
  
**public class** ProAndCons {  
 **public static void** main(String[] args) {  
 Resource r = **new** Resource();  
 Producer pro = **new** Producer(r);  
 Consumer con = **new** Consumer(r);  
 Thread t1 = **new** Thread(pro);  
 Thread t2 = **new** Thread(pro);  
 Thread t3 = **new** Thread(con);  
 Thread t4 = **new** Thread(con);  
  
 t1.start();  
 t2.start();  
 t3.start();  
 t4.start();  
 }  
}

```

两个线程间的协同操作 使用while保证每一次线程被唤醒之后都需要进行判断，不会发生两个线程同时进行wait（）发生死锁问题

#### 死锁

产生死锁的条件

1. 至少一个资源不共享
2. 至少一个任务持有一个资源并且等待获取另一个被别的任务持有的资源
3. 资源不能任务抢占
4. 必须循环等待

只需要破坏其中一个就不会产生死锁

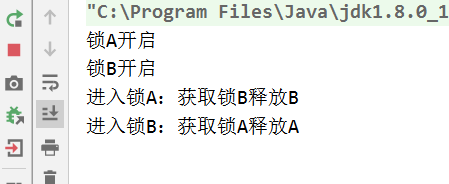
1. 两个线程发生交叉闭环 线程A 获得锁A需要锁B 才能释放A ，线程B 获得锁B的情况下才能释放A，等待对方释放资源自己才能结束陷入死锁

```javascript

**package** main;  
**class** DeadLockA **implements** Runnable{  
 @Override  
 **public void** run() {  
 **try**{  
 System.***out***.println(**"锁A开启"**);  
 }**catch** (Exception e){  
 **while**(**true**){  
 **synchronized** (DeadLock.***obj1***){  
 System.***out***.println(**"进入锁A：获取锁B释放B"**);  
 **synchronized** (DeadLock.***obj2***){  
 System.***out***.println(**"进入锁A的锁B区成功"**);  
 }  
 }  
 }  
 }  
 }  
}  
**class** DeadLockB **implements** Runnable{  
 @Override  
 **public void** run() {  
 **try**{  
 System.***out***.println(**"锁B开启"**);  
 }**catch** (Exception e){  
 **while**(**true**){  
 **synchronized** (DeadLock.***obj2***){  
 System.***out***.println(**"进入锁B：获取锁A释放A"**);  
 **synchronized** (DeadLock.***obj1***){  
 System.***out***.println(**"进入锁B的锁A区成功"**);  
 }  
 }  
 }  
 }  
 }  
}  
**public class** DeadLock {  
 **public static final** String ***obj1*** = **"锁A"**;  
 **public static final** String ***obj2*** = **"锁B"**;  
  
 **public static void** main(String[] args) {  
 DeadLockA a = **new** DeadLockA();  
 DeadLockB b = **new** DeadLockB();  
 Thread t1 = **new** Thread(a);  
 Thread t2 = **new** Thread(b);  
 t1.start();  
 t2.start();  
 }  
}

```

结果



线程A 等待线程B 线程B等待线程A 发生死锁问题

##### 类集

1. ArrayList Vctor LinkedList

|  |  |  |
| --- | --- | --- |
| ArrayList | Vctor | LinkedList |
| 数组实现 | 数组实现 | 带头双向链表 |
| 数组扩容1.5倍 | 数组扩容2倍 capactiyIncremment设置扩容大小 大于0 数值 小于零默认两倍 | 不需要扩容 增加节点 |
| 插入删除时间复杂度O（1）头尾  数组内O（n） | 插入删除时间复杂度O（1）头尾  数组内O（n） | 检索一个元素时间复杂度O（n） |

1. fail-fast与fail-safe
2. 出现同步修改 当一个或多个线程遍历一个集合 此时另一个线程修改内容出现并发修改
3. Fail-fast机制

fail-fast遍历一个集合时当结构被修改抛出Concurrent Modification Exception

1. 单线程环境 集合被创建在遍历过程中修改结构
2. 多线程环境 一个线程遍历一个线程修改
3. Fail-fast如何检测

迭代器遍历过程中直接访问内部数据，为了保证遍历过程中不被修改，迭代器内部为元素添加modeCount标记 当执行hashNext()/next()遍历下一个元素之前都会检测modeCount变量是否等于expectedmodeCount 相等遍历不相等抛出异常终止遍历

1. java.util包下的集合类都是快速失败的，不能在多线程下发生并发修改（迭代过程中被修改）
2. Fail-safe机制

在遍历时 拷贝原有集合元素 遍历拷贝集合

遍历拷贝集合所以在进行迭代器检测时不能检测到

java.util.concurrent包下的容器都是安全失败

1. Set与Map
2. List Set Map 区别

|  |  |  |
| --- | --- | --- |
| List | Set | Map |
| 接口 继承Collection | 接口 继承Collection | 接口 |
| ArrayLsit 数组 不安全  Vector 数组 安全  linkedList 链表 不安全 | HashSet哈希表 元素无序唯一  LinkedHashSet 链表+哈希表 有序唯一  TreeSet 红黑树 唯一 有序 | HashMap 线程不安全  TreeMap 有序  HashTable 线程安全  LinkedHashMap |
| 数组类似动态增长 查找元素效率高 插入删除元素效率低 | 检索效率低 删除插入效率高  存放数据不重复 | 适合存放键值对数据 |
| LinkedList ArrayList HashSet非线程安全  Vector线程安全 | 底层使用map集合 | HashMap非线程安全  HashTable线层安全 |

1. HashMap、TreeMap、Hashtable

|  |  |  |
| --- | --- | --- |
| HashMap | TreeMap | Hashtable |
| 哈希表 继承abstractMap | 基于红黑树 key排序  实现SortedMap接口 | 哈希结构 |
| 允许null键null值 |  | 不能存入null键null值 |
| 线程不安全 JDK1.2 |  | 线程安全JDK1.0 |

1. HashMap特性

HashMap可以存入 null键和值，HashTable不能 hashMap线程不安全效率高，hastable线程安全效率低

1. HashMap工作原理，get方法工作原理

基于hashing（哈希法）原理实现。使用put存储到对象到hashmap中当我们使用put传递key和value时，使用get从hashmap中获取对象

1. 先调用hashCode（）方法 返回hashCode用于找到bucket位置来存储一个或者多个Entry对象（key,value,next）
2. HashMap实在bucket中存储key和value作为Map.Entry
3. 当两个对象hashCode相同是发生什么

存放：HashCode相同 所以bucket位置相同 会发生碰撞 HashMap使用链表存储所以Entry存储到链表中

获取：找到bucket为重 调用keys.equals发放去找链表正确的节点 最终找到值

1. 如果hashMap大小超过负载因子

默认负载因子 大小0.75 当一个map填满75%时 会自动库容原来大小两倍bucket数组，来重新调整map大熊 并将原来对象放入新的bucket数组中 整个过程rehashing

HashMap源码实现(参考https://blog.csdn.net/danielzhou888/article/details/84332201)

```javascript

**package** main;  
  
**public class** CustomHashMap<K,V> {  
 **private class** Entry<K,V>{  
 **int hash**;  
 K **key**;  
 V **value**;  
 Entry<K,V> **next**;  
  
 Entry(**int** hash,K key,V value,Entry<K,V> next){  
 **this**.**hash** = hash;  
 **this**.**key** = key;  
 **this**.**value** = value;  
 **this**.**next** = next;  
 }  
 }  
 **private static final int *DEFAULT\_CAPATITY*** = 1<<4;  
  
 **private** Entry<K,V>[] **table**;  
  
 **private int capacity**;  
  
 **private int size**;  
  
 **public** CustomHashMap(){  
 **this**(***DEFAULT\_CAPATITY***);  
 }  
  
 **public** CustomHashMap(**int** capacity){  
 **if**(capacity<0){  
 **throw new** IllegalArgumentException();  
 }**else**{  
 **table** = **new** Entry[capacity];  
 **size** = 0;  
 **this**.**capacity** = capacity;  
 }  
 }  
  
 **public int** size(){  
 **return size**;  
 }  
  
 **public boolean** isEmpty(){  
 **return size** == 0?**true**:**false**;  
 }  
  
 **private int** hash(K key){  
 **double** tmp = key.hashCode()\*(Math.*pow*(5,0.5)-1)/2;  
 **double** digit = tmp - Math.*floor*(tmp);  
 **return** (**int**)Math.*floor*(digit\***capacity**);  
 }  
  
 **public void** put(K key,V value){  
 **if**(key == **null**){  
 **throw new** IllegalArgumentException();  
 }  
 **int** hash = hash(key);  
 Entry<K,V> nEntry = **new** Entry<K,V>(hash,key,value,**null**);  
 Entry<K,V> entry = **table**[hash];  
 **while**(entry!=**null**){  
 **if**(entry.**key**.equals(key)){  
 entry.**value** = value;  
 **return** ;  
 }  
 entry = entry.**next**;  
 }  
 nEntry.**next** = **table**[hash];  
 **table**[hash] = nEntry;  
 **size**++;  
 }  
  
 **public** V get(K key){  
 **if**(key == **null**){  
 **throw new** IllegalArgumentException();  
 }  
 **int** hash = hash(key);  
 Entry<K,V> entry = **table**[hash];  
 **while**(entry!=**null**){  
 **if**(entry.**key**.equals(key)){  
 **return** entry.**value**;  
 }  
 entry = entry.**next**;  
 }  
 **return null**;  
 }  
  
 **public static void** main(String[] args) {  
 CustomHashMap<String,String> map = **new** CustomHashMap<String,String>();  
 map.put(**"1"**,**"2"**);  
 map.put(**"1"**,**"22"**);  
 System.***out***.println(map.get(**"1"**));  
 }  
}

```

#### Juc

1. 并发并行
2. 并发多个任务交替进行
3. 并行真正意义上同时进行
4. 避免死锁的方式
5. 避免一个线程同时获得多个锁
6. 避免一个线程内部占有多个资源，保证一个所一个资源
7. 尝试使用定时锁
8. 数据库锁 加锁解锁一个数据库连接内

（2）线程的几种状态

1. wait() join() LockSupport.Lock() 线程进入WAITING
2. wait(long timeout)，sleep(long),join(long),LockSupport.parkNanos(),LockSupport.parkUtil()进入TIMED\_WAITING状态
3. 当超时等待时间到达 线程切换到Runable 也可以通过notify() notifyAll() 是线程进入Runable状态
4. 线程出现资源竞争时，即等待获取锁的时候，线程会进入到BLOCKED阻塞状态
5. 线程获取锁时，线程进入到Runable状态
6. 线程运行结束后，线程进入到TERMINATED状态
7. 线程进入到synchronized方法或者synchronized代码块时，线程切换到的是BLOCKED状态使用java.util.concurrent.locks下lock进行加锁的时候线程切换的是WAITING或者TIMED\_WAITING状态，因为lock会调用LockSupport的方法。

|  |  |
| --- | --- |
| 状态 |  |
| NEW | 初始状态 |
| RUNNABLE | 运行状态 |
| BLOCKED | 阻塞状态 |
| WAITING | 等待状态 |
| TIME\_WAITING | 超市等待 指定时间返回 |
| TERMINATED | 中止状态 |

1. 线程方法
2. interrupted 线程中断 一个线程一个标志位 抛出****InterruptedException清除标志位如果标志为true当前线程不继续续 只有为false才能继续向下****
3. Join 线程等待 一个线程实例A执行了threadB.join()当前线程A会等待threadB线程终止后threadA才会继续执行
4. Sleep 线程休眠 不会失去锁

Sleep与wait区别

1. sleep是Thread静态方法 wait是Object实例方法
2. Wait() 必须在同步方法或者同步块中得拥有对象锁 会释放对象 sleep随时调用不会释放对象锁
3. Sleep到达之后继续 wait(）等待notifty才能被唤醒
4. yeild当前线程让出cpu 让出的时间片只会分配****给当前线程相同优先级****的线程。

Sleep与yield

1. sleep交出时间片其他线程都都可以竞争
2. yield只允许当前线程相同优先级能够获取时间片

5 .守护线程

Daemon 如 垃圾回收 jit

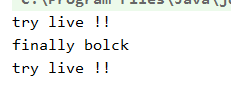
****守护线程在退出的时候并不会执行finnaly块中的代码****

****```javascript****

**public static void** main(String[] args) {  
 Thread daemonThread = **new** Thread(**new** Runnable() {  
 @Override  
 **public void** run() {  
 **while**(**true**){  
 **try** {  
 System.***out***.println(**"try live !!"**);  
 Thread.*sleep*(500);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }**finally** {  
 System.***out***.println(**"finally bolck"**);  
 }  
 }  
 }  
 });  
  
 daemonThread.setDaemon(**true**);  
 daemonThread.start();  
  
 **try** {  
 Thread.*sleep*(800);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
}

```

运行结果



在执行过程中mian线程执行完毕 但是daemonThread.setDaemon(true); 让daemonThread获得一次时间片机会 正常打印try中的一次 守护线程退出时不会执行finnaly中的代码

#### JMM（java内存模型）

（1）多线程共享变量

****实例域，静态域和数组元素****都是放在堆内存中（所有线程均可访问到，是可以共享的）

局部变量，方法定义参数和异常处理器参数不会在线程间共享

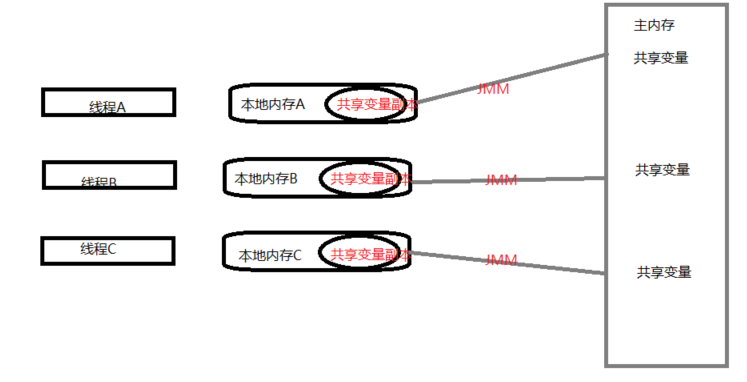
（2）JMM结构模型

1. 线程A将主内存共享变量读取读入线程A的工作内存后并进行操作，之后将数据重 新写回到主内存中

2. 线程B从主存中读取最新的共享变量

如果线程A更新后没有及时写会主内存而此时线程B读到的是过期的数据出现了“脏读”现象

可以通过同步机制或者volatile关键字让共享变量强制刷新到内存 变量具有可见性



（2）重排序

为了提高性能，编译器和处理器常常会对指令进行重排序

1. 编译器优化的重排序。
2. 指令级并行的重排序
3. 内存系统的重排序

as-if-serial不管怎么重排序（编译器和处理器为了提供并行度）（单线程）程序的执行结果不能被改变。

1. volatile保持内存可见性和防止指令重排序

（3）happens-before

JMM可以通过happens-before关系向程序员提供跨线程的内存可见性保证

如果A线程的写操作a

B线程的读操作b

存在happens-before关系，尽管a操作和b操作在不同的线程中执行，但JMM向程序员保证a操作将对b操作可见

1. 如果一个操作happens-before另一个操作，那么第一个操作的执行结果将对第二个操作可见，而且第一个操作的执行顺序排在第二个操作之前。
2. 两个操作之间存在happens-before关系，并不意味着Java平台的具体实现必须要按照happens-before关系指定的顺序来执行。

|  |  |
| --- | --- |
| as-if-serial | happens-before |
| 单线程内程序的执行结果不被改变 | 多线程程序的执行结果不被改变。 |
| 尽可能单线程程序是按程序的顺序来执行的 | 尽可能正确同步的多线程程序是按happens-before指定的顺序来执行的。 |

1. as-if-serial语义和happens-before这么做的目的，都是为了在不改变程序执行结果的前提下，尽可能地提高程序执行的并行度。

（4）happens-before具体规则

· 程序顺序规则：一个线程中的每个操作，happens-before于该线程中的任意后续操作。

· 监视器锁规则：对一个锁的解锁，happens-before于随后对这个锁的加锁。

· volatile变量规则：对一个volatile域的写，happens-before于任意后续对这个volatile域的读。

· 传递性：如果A happens-before B，且B happens-before C，那么A happens-before C。

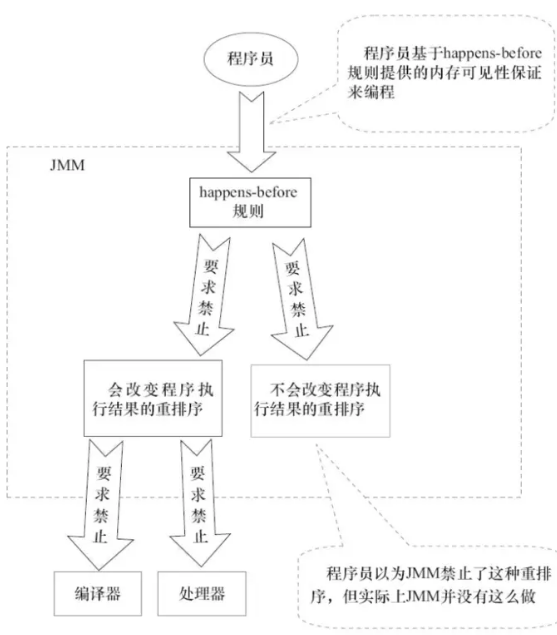
· start()规则：如果线程A执行操作ThreadB.start()（启动线程B），那么A线程的ThreadB.start()操作happens-before于线程B中的任意操作。

· join()规则：如果线程A执行操作ThreadB.join()并成功返回，那么线程B中的任意操作happens-before于线程A从ThreadB.join()操作成功返回。

· 程序中断规则：对线程interrupted()方法的调用先行于被中断线程的代码检测到中断时间的发生。

· 对象finalize规则：一个对象的初始化完成（构造函数执行结束）先行于发生它的finalize()方法的开始。

1. 总结



1. 对于会改变程序执行结果的重排序，JMM要求编译器和处理器必须禁止这种重排序。
2. 对于不会改变程序执行结果的重排序，JMM对编译器和处理器不做要求（JMM允许这种 重排序）

#### Syncharonized

1. 适用范围
2. 方法

```javascript

Public synchronized void nethod(){

}

Public static synchronized void menthod(){

}

```

1. 代码块

```javascript

Sycnronized (this){

}

Sychronized(class.class){

}

```

在实例方法中锁的该类的实例对象 静态方法该类的对象

代码块中锁当前对象，该类的class反射对象

1. Monitor机制
2. 执行同步代码块后首先要先执行****monitorenter****指令，退出的时候****monitorexit****指令

2. 使用Synchronized进行同步，其关键就是必须要对对象的监视器monitor进行获取

3. 当线程获取monitor后才能继续往下执行，否则就只能等待。（获取的过程互斥）

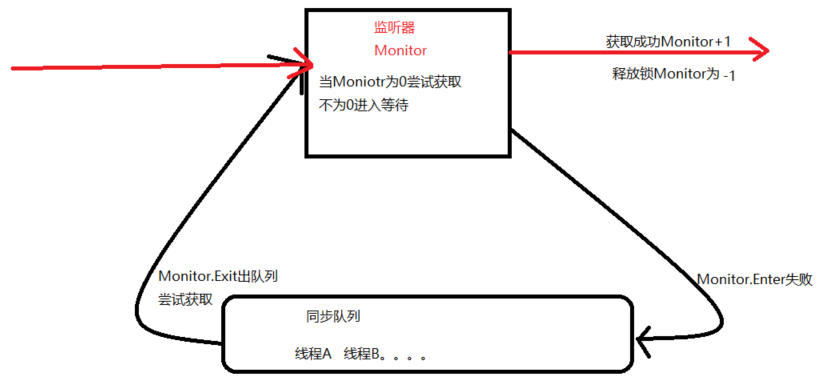
4. 同一时刻只能有一个线程获取monitor

5. 同一锁程中，线程不需要再次获取同一把锁。Synchronized先天具有重入性

6. 每一个线程进入代码块都会尝试获取monitor如果monitor不为0则等待

7. 当线程成功获取锁对象monitor+1 释放锁之后-1

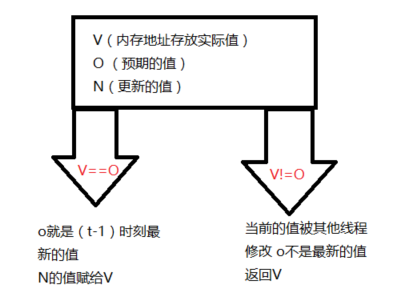
8. 每个对象拥有一个计数器，当线程获取该对象锁后，计数器就会加一，释放锁后就会将计数器减一。



1. CAS
2. 线程获取锁是一种****悲观锁策略****假设每一次执行临界区代码都会产生冲突，所以当前线程获取到锁的 时候同时也会阻塞其他线程获取该锁。

CAS成为无所操作 乐观策略 假设所有线程访问共享资源不会出现冲突

1. 操作过程



1. 产生的ABA问题

线程A将旧值A变为了成B线程B将再变成A 线程A在进入发现没有改变实际上已经改变

解决方案

添加标记 版本号

1. 自旋时间过长 不会将线程挂起CPU跑无用指令做下一次尝试，产生的资源消耗
2. 只能保证一个共享变量原子操作

（4）优化